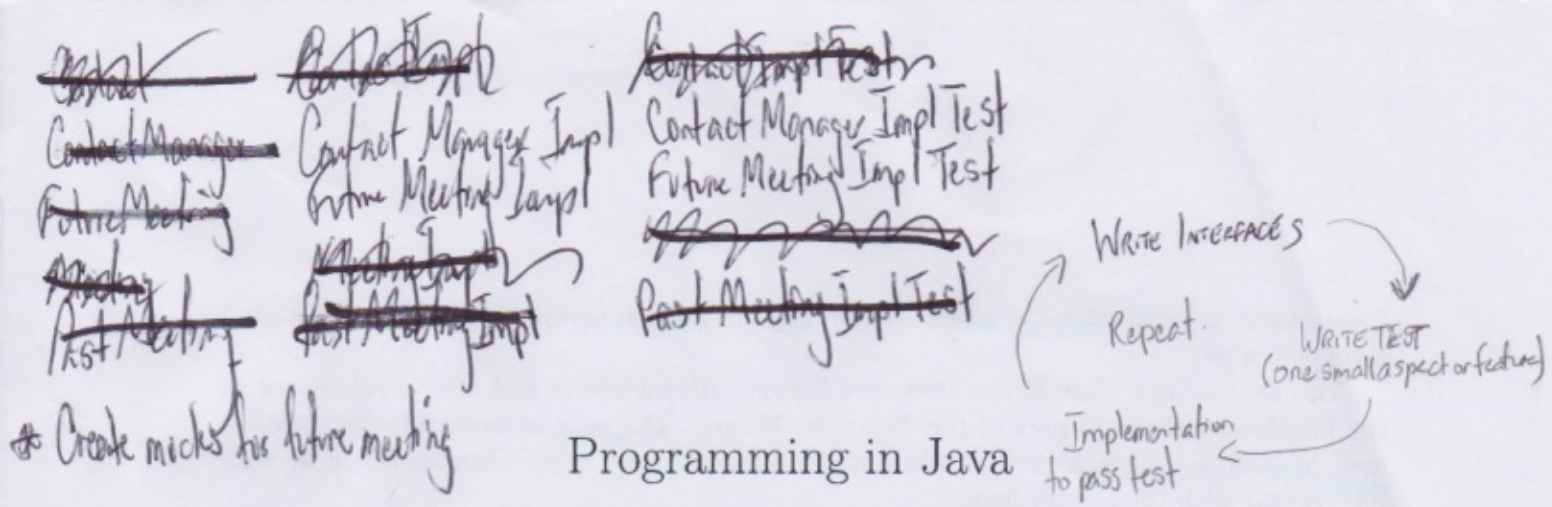

Programming in Java

Coursework Three

2015-16

Code Viscosity: When the automated tests verify that everything is still working — even those parts of the program unknown to the programmer — one can be confident that the last change did not break anything.

See the Moodle site for further information concerning submission dates.

Your `github` repo should be named "`cw-cm`".

1 Introduction

Launching a start-up company is difficult. Apart from creating a great product, you must talk to a lot of people to get them interested in the project: clients, venture capital, government agencies... it is really difficult to keep track of everybody!

When is your next meeting? Who will you meet? What did they say the last time you talked to them?

The purpose of this assignment it writing a program to keep track of contacts and meetings. The application will keep track of contacts, past and future meetings, etc. When the application is closed, all data must be stored in a text file called contacts.txt . This file must be read at startup to recover all data introduced in a former session (the format of the file if up to you: you can use XML, comma-separated values (CSV), or any other format).

2 Goals, methodology, and conventions

The program should implement the interfaces provided below in Section 3. The functionality should be as described. No exceptions can be thrown except in the situations described on the interfaces.

Classes that implement an interface should follow the convention of having the same name as the interface with the Impl suffix unless instructed otherwise. For example, if there is only one class that implements ContactManager, it must be called ContactManagerImpl. Be careful with this, as it may interfere with automatic tools that will analyse your code (e.g. automatic testing).

Unless instructed otherwise, all classes must have only one constructor with no parameters.

In order to complete this piece of coursework, you must follow the Test Driven Development (TDD) methodology. That is, you must first write the interfaces (most of them are already provided below), then write the test for one small aspect or feature, then write the implementation that passes the test, then repeat. Your commit history should clearly show that you wrote the tests before the implementation and not after, and that you added new

functionality in small cycles of interface–test–implementation and not big chunks of untested code in one go.

You can use any class of the Java core library. All interfaces and classes referenced in the interfaces below are part of the Java core library. The only external library that you need is JUnit4 (and HamCrest). You may be allowed to use other libraries if you so wish but consult with the faculty first.

3 Interfaces

3.1 ContactManager

```
import java.util.Calendar;
import java.util.List;
import java.util.Set;

/**
 * A class to manage your contacts and meetings.
 */
public interface ContactManager {
    /**
     * Add a new meeting to be held in the future.
     *
     * An ID is returned when the meeting is put into the system. This
     * ID must be positive and non-zero.
     *
     * @param contacts a list of contacts that will participate in the meeting
     * @param date the date on which the meeting will take place
     * @return the ID for the meeting
     * @throws IllegalArgumentException if the meeting is set for a time
     *       in the past, of if any contact is unknown / non-existent.
     * @throws NullPointerException if the meeting or the date are null
    */
    int addFutureMeeting(Set<Contact> contacts, Calendar date);

    /**
     * Returns the PAST meeting with the requested ID, or null if it there is none.
     *
     * The meeting must have happened at a past date.
     *
     * @param id the ID for the meeting
     * @return the meeting with the requested ID, or null if it there is none.
     * @throws IllegalStateException if there is a meeting with that ID happening
     *         in the future
     */
    PastMeeting getPastMeeting(int id);

    /**
     * Returns the FUTURE meeting with the requested ID, or null if there is none.
     *
     * @param id the ID for the meeting
     * @return the meeting with the requested ID, or null if it there is none.
```

```
     * @throws IllegalArgumentException if there is a meeting with that ID happening
     *         in the past
     */
    FutureMeeting getFutureMeeting(int id);

    /**
     * Returns the meeting with the requested ID, or null if it there is none.
     *
     * @param id the ID for the meeting
     * @return the meeting with the requested ID, or null if it there is none.
     */
    Meeting getMeeting(int id);

    /**
     * Returns the list of future meetings scheduled with this contact.
     *
     * If there are none, the returned list will be empty. Otherwise,
     * the list will be chronologically sorted and will not contain any
     * duplicates.
     *
     * @param contact one of the users contacts
     * @return the list of future meeting(s) scheduled with this contact (maybe empty).
     * @throws IllegalArgumentException if the contact does not exist
     * @throws NullPointerException if the contact is null
     */
    List<Meeting> getFutureMeetingList(Contact contact);

    /**
     * Returns the list of meetings that are scheduled for, or that took
     * place on, the specified date
     *
     * If there are none, the returned list will be empty. Otherwise,
     * the list will be chronologically sorted and will not contain any
     * duplicates.
     *
     * @param date the date
     * @return the list of meetings
     * @throws NullPointerException if the date are null
     */
    List<Meeting> getMeetingListOn(Calendar date);

    /**
     * Returns the list of past meetings in which this contact has participated.
     *
     * If there are none, the returned list will be empty. Otherwise,
     * the list will be chronologically sorted and will not contain any
     * duplicates.
     *
     * @param contact one of the users contacts
     * @return the list of future meeting(s) scheduled with this contact (maybe empty).
     * @throws IllegalArgumentException if the contact does not exist
     * @throws NullPointerException if the contact is null
     */
    List<PastMeeting> getPastMeetingListFor(Contact contact);
```

```
/**
 * Create a new record for a meeting that took place in the past.
 *
 * @param contacts a list of participants
 * @param date the date on which the meeting took place
 * @param text messages to be added about the meeting.
 * @throws IllegalArgumentException if the list of contacts is
 *
 empty, or any of the contacts does not exist
 * @throws NullPointerException if any of the arguments is null
 */
void addNewPastMeeting(Set<Contact> contacts, Calendar date, String text);

/**
 * Add notes to a meeting.
 *
 * This method is used when a future meeting takes place, and is
 * then converted to a past meeting (with notes) and returned.
 *
 * It can be also used to add notes to a past meeting at a later date.
 *
 * @param id the ID of the meeting
 * @param text messages to be added about the meeting.
 * @throws IllegalArgumentException if the meeting does not exist
 * @throws IllegalStateException if the meeting is set for a date in the future
 * @throws NullPointerException if the notes are null
 */
PastMeeting addMeetingNotes(int id, String text);

/**
 * Create a new contact with the specified name and notes.
 *
 * @param name the name of the contact.
 * @param notes notes to be added about the contact.
 * @return the ID for the new contact
 * @throws IllegalArgumentException if the name or the notes are empty strings
 * @throws NullPointerException if the name or the notes are null
 */
int addNewContact(String name, String notes);

/**
 * Returns a list with the contacts whose name contains that string.
 *
 * If the string is the empty string, this methods returns the set
 * that contains all current contacts.
 *
 * @param name the string to search for
 * @return a list with the contacts whose name contains that string.
 * @throws NullPointerException if the parameter is null
 */
Set<Contact> getContacts(String name);

/**
```

```
     * Returns a list containing the contacts that correspond to the IDs.
     * Note that this method can be used to retrieve just one contact by passing only one ID.
     *
     * @param ids an arbitrary number of contact IDs
     * @return a list containing the contacts that correspond to the IDs.
     * @throws IllegalArgumentException if no IDs are provided or if
     *     any of the provided IDs does not correspond to a real contact
     */
    Set<Contact> getContacts(int... ids);

    /**
     * Save all data to disk.
     *
     * This method must be executed when the program is
     * closed and when/if the user requests it.
     */
    void flush();
}
```

3.2 Contact

The implementation of this interface must have two constructors. The most general constructor must have three parameters: int, String, String. The first one corresponds to the ID provided by the ContactManager, the next one corresponds to the name, and the last one corresponds to the initial set of notes about the contact. Another, more restricted constructor must have two parameters only: ID and name. If the ID provided is zero or negative, a `IllegalArgumentException` must be thrown. If any of the references / pointers passed as parameters to the constructor is null, a `NullPointerException` must be thrown.

```
/**
 * A contact is a person we are making business with or may do in the future.
 *
 * Contacts have an ID (unique, a non-zero positive integer),
 * a name (not necessarily unique), and notes that the user
 * may want to save about them.
 */
public interface Contact {
    /**
     * Returns the ID of the contact.
     *
     * @return the ID of the contact.
     */
    int getId();

    /**
     * Returns the name of the contact.
     *
     * @return the name of the contact.
     */
    String getName();
```

```
    /**
     * Returns our notes about the contact, if any.
     *
     * If we have not written anything about the contact, the empty
     * string is returned.
     *
     * @return a string with notes about the contact, maybe empty.
     */
    String getNotes();

    /**
     * Add notes about the contact.
     *
     * @param note the notes to be added
     */
    void addNotes(String note);
}
```

3.3 Meeting

The class implementing this interface must be *abstract*. It must have only one constructor with three parameters: an ID (`int`), a date, and a set of contacts that must be non-empty (otherwise, an `IllegalArgumentException` must be thrown). A `IllegalArgumentException` must also be thrown in the case the ID provided was non-positive or zero. If any of the references / pointers passed as parameters is null, a `NullPointerException` must be thrown.

```
import java.util.Calendar;
import java.util.Set;
/**
 * A class to represent meetings
 *
 * Meetings have unique IDs, scheduled date and a list of participating contacts
 */
public interface Meeting {
    /**
     * Returns the id of the meeting.
     *
     * @return the id of the meeting.
     */
    int getId();

    /**
     * Return the date of the meeting.
     *
     * @return the date of the meeting.
     */
    Calendar getDate();

    /**
     * Return the details of people that attended the meeting.
     *
     * The list contains a minimum of one contact (if there were
```

```
     * just two people: the user and the contact) and may contain an
     * arbitrary number of them.
     *
     * @return the details of people that attended the meeting.
     */
    Set<Contact> getContacts();
}
```

3.4 PastMeeting

The class implementing this interface must have only one constructor with four parameters: an ID (`int`), a date, a set of contacts that must be non-empty (otherwise, an `IllegalArgumentException` must be thrown), and a String containing the notes for the meeting. If any of the references / pointers passed as parameters is null, a `NullPointerException` must be thrown.

```
/**
 * A meeting that was held in the past.
 *
 * It includes your notes about what happened and what was agreed.
 */
public interface PastMeeting extends Meeting {
    /**
     * Returns the notes from the meeting.
     *
     * If there are no notes, the empty string is returned.
     *
     * @return the notes from the meeting.
     */
    String getNotes();
}
```

3.5 FutureMeeting

The class implementing this interface must have only one constructor with three parameters: an ID (`int`), a date, and a set of contacts that must be non-empty (otherwise, an `IllegalArgumentException` must be thrown). If any of the references / pointers passed as parameters is null, a `NullPointerException` must be thrown.

```
/**
 * A meeting to be held in the future
 */
public interface FutureMeeting extends Meeting{
    // No methods here, this is just a naming interface
    // (i.e. only necessary for type checking and/or downcasting)
}
```

4 Submission and grading

The assignment will be automatically cloned on the date specified on the Moodle site at 23h59.59 London time. You are encouraged to leave everything ready well in advance —both the programming and pushing it to your GitHub account— to avoid last-minute problems (e.g. GitHub may be down on that weekend for maintenance). If the code is not available at the GitHub account by the deadline, the assignment will not be marked.

The assignment will be graded according to its compliance with the provided specification; the simplicity, clarity, and generality of the code (including succint but illustrative comments and JavaDoc); and the compliance with good practices of version control (e.g. commiting often and in small pieces, use of descriptive commit messages, committing only source code and not binary or class files).

Regardless of the times you choose to push your changes to GitHub, you should commit early and often. In case of suspected plagiarism, your version control history will be used as additional evidence to judge the case (including partial clones that may be made in dates before the deadline). It is in your best interest to commit very often (and to use adequate commit messages) to make it clear that the process of creation is entirely your own.